01 Jan 1982

# Applying Software Complexity Metrics to Program Maintenance

Kenneth Magel

Raymond Michael Kluczny
*Missouri University of Science and Technology*, rkluczny@mst.edu

Warren A. Harrison

Arlan R. Dekock
*Missouri University of Science and Technology*, adekock@mst.edu



## Recommended Citation

Magel, K., Kluczny, R. M., Harrison, W. A., & Dekock, A. R. (1982). Applying Software Complexity Metrics to Program Maintenance. *Computer* Institute of Electrical and Electronics Engineers (IEEE).
The definitive version is available at https://doi.org/10.1109/MC.1982.1654138

*Predicting software complexity can save millions in maintenance costs, but while current measures can be used to some degree, most are not sufficiently sensitive or comprehensive.*

# Applying Software Complexity Metrics to Program Maintenance

**Warren Harrison, Kenneth Magel, Raymond Kluczny, and Arlan DeKock**
**University of Missouri-Rolla**

Over the past several years, computer scientists have devoted a great deal of effort to measuring computer program "complexity," since many large software systems can be used for 10, 15, or even 20 years. A large part of that time involves maintenance activities, which include all changes made to a piece of software after it has been delivered to and accepted by the final user. Consequently, maintenance is most affected by program complexity.

Recent estimates suggest that about 40 to 70 percent of annual software expenditures involve maintenance of existing systems. Clearly, if complexities could somehow be identified, then programmers could adjust maintenance procedures accordingly. What is needed is some method of pinpointing the characteristics of a computer program that are difficult to maintain and measuring the degree of their presence (or lack of it). Such a method could be used in preparing "quality specifications" for programs that are to be written; checking specification compliance of programs after they have been written, but before they are delivered; making proper design trade-offs between development and maintenance costs; and selecting a particular type of software.

## Software complexity

The degree to which characteristics that impede software maintenance are present is called software *maintainability* and is driven primarily by *software complexity,* the measure of how difficult the program is to comprehend and work with. Maintenance characteristics that are affected by complexity include software understandability, software modifiability, and software testability.

Various approaches may be taken in measuring complexity characteristics, such as Baird and Noma's[1] approach, in which scales of measurement are divided into the following four types:

(1) *Nominal scales.* The measure classifies the items. For example, programs are grouped into classifications of "not difficult to understand," "moderately difficult to understand," "difficult to understand," and "very difficult to understand."

(2) *Ordinal scales.* The measure actually ranks individual items. For example, we would say not only that program *A,* program *B,* and program C are all "moderately difficult to understand," but also that program *B* was more difficult to understand than program *C,* and program *A* was more difficult to understand than program *B.*

(3) *Interval scales.* The measure not only ranks items in relation to each other but also tells how far apart they are. For example, we would say not only that program *A* was more difficult than program *B* but also that program *A* was 10 "units of difficulty" more difficult to understand than program *B.*

4) *Ratio scales.* The measure not only ranks the items and determines how far apart they are from each other but also determines how far the measures lie from a total lack of the characteristic being measured. This allows multiplication and division to be used on the resulting measures, so we can obtain measures that indicate that program *A* is twice as difficult to understand as program *B.* This property is unique to the ratio scale.

The most flexible type of measurement seems to be the ratio scale, but we would be hard put to find a degree of impeding characteristics that even approaches zero complexity, so this scale is not really the most effective. On the other hand, an ordinal scale would be feasible, and in some cases, even an interval scale could be developed, but a major problem with interval scales is determining the "unit of difficulty" and its meaning.

We believe the ordinal scale to be the best choice for examining complexity metrics, and all measures discussed in this article are in that framework.


0018-9162/82/0900-0065$00.75 

## Existing complexity measures

Because much of the work on complexity metrics has been done in the last five years, many different methods are being used. Basili[2] has suggested that program size, data structures, data flow, and flow of control can affect maintenance. A number of measures have been developed to evaluate each of these characteristics, and several hybrid measures have been developed to consider more than one simultaneously.

**Program size.** The most straightforward approach is based on program size, and as Elshoff[3] has pointed out, very large programs incur problems just by virtue of the volume of information that must be absorbed to understand the program. Program size is easy to calculate, is widely applicable, and has definable measures, two of which are lines of code and Halstead's software science.

The most common form of measuring program size is by simply counting the lines of code. Unfortuntely, not everyone agrees on what makes up a line of code, and the question remains whether only executable source statements, executable source statements and data declaration statements, or all statements including comments should be part of the measure.

Halstead's software science[4] is based on a refinement of measuring program size by counting lines of code. It is one of the most widely accepted measures in industry and universities and has been supported by several empirical studies.[5-11]. Halstead's metrics measure the number of unique operators $n1$, the number of unique operands $n2$, the total number of operators $N1$, and the total number of operands $N2$. From these measures Halstead defines the vocabulary of the program as $n = n1 + n2$ and the total length of the program as $N = N1 + N2$. He further computes a predicted program length as $N' = n1 \log2 n1 + n2 \log2 n2$. The calculation of the predicted program length is called length equation and is supported by the empirical results of Halstead,[4] Bohrer,[5] and Elshoff.[6]

Halstead computes program volume as $V = N \log2 n$ and minimal or potential volume as $V* = n* \log2 n*$, where $n*$ is the size of the potential vocabulary. The potential vocabulary is that needed to invoke a "built-in" function (if one exists) to perform the desired task. For example, the potential implementation of a sort procedure might look like call sort $(x,n)$, where $x$ is the array to be sorted and $n$ is the number of elements in the array. The potential vocabulary would include call, sort(. . .), $x$ and $n$. Therefore, $n* = 4$.

The program level is subsequently defined as the ratio of potential volume to actual volume and is computed as $L = V*/V$. An approximation of program level is $L' = 2/n1 \times n2/N2$. Halstead found that the approximation has a correlation coefficient of 0.90 to the actual observed value.[4] Clearly, the larger the volume of the existing program relative to the volume of the potential program, the lower the program level.

Halstead uses the volume and program level to calculate the intelligence content of the program, expressed as $I = L' \times V$. He indicates that this relationship correlates best with total programming and debugging time and that intelligence content is a likely candidate for a complexity measure. However, the intelligence content of a program appears to remain invariant under translation from one programming language to another and increases only as the complexity of the problem increases.

Halstead uses the volume and program level to measure the effort required to generate a piece of software by $E = V/L$. Funami and Halstead[7] point out that $E$ has often been used to measure the effort required to comprehend an implementation. For this reason, they suggest that $E$ may be used as a measure of program clarity. The use of Halstead's metrics on a simple bubble sort, adapted from an example used by Fitzsimmons and Love,[12] is shown in Figures 1 and 2 and Tables 1 through 5.

In general, metrics based on measures of program size have been the most successful to date, with experimental evidence indicating that larger programs have greater maintenance costs than smaller ones. However, although program size metrics tend to work well in ranking programs with widely varying sizes, other characteristics such as data structures, data flow, and flow of control become vitally important as the size difference decreases. In other

```
(a)  BEGIN
         DO I=2 TO N;
             DO J=1 TO I;
                 IF X(I) < X(J) THEN DO;
                                     SAVE=X(I);
                                     X(I)=X(J);
                                     X(J)=SAVE;
                                 END;
             END;
         END;
     END;

(b)  CALL SORT (X,N)
```

**Figure 1. Actual (a) and potential (b) implementations of a bubble sort program.**

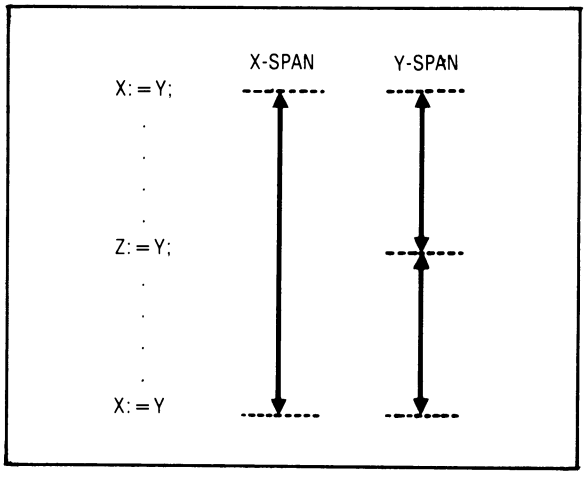



**Figure 2. Span between data references.**

words, program size metrics can be a very good nominal scale to use in putting programs into one "complexity category," but it may not be able to distinguish between different programs within the same category.

**Data structures and data flow.** Another factor that influences software complexity is the configuration and use of data within the program. Several methods can be used to measure complexity by the way program data are used, organized, or allocated.

*Span between data references.* This technique, which is based on the locality of data references within the program, has no supporting empirical studies to demonstrate its correlation with maintenance, but it is intuitively appealing.

A *span* is the number of statements between two references to the same identifier with no intervening references to that identifier. Consequently an identifier has $n-1$ spans for an identifier that occurs $n$ times in the source code (Figure 2).

Elshoff[3] found that of 120 production programs used at General Motors Corporation, over 13 percent had data reference spans of 100 statements or more. Maintenance activities might require a programmer to determine what value a variable has at a particular point, and in more than one case in eight the programmer would have to search through at a level of 100 statements with Elshoff's data.

*Segment-global usage pair.* This measure, discussed by Basili[2,13] bases program complexity on the use of global data within the program. It is useful for large programs that consist of several modules or segments, but again no empirical studies have been reported.

A segment-global usage pair $(p,r)$ is used to signify the instance of a segment $p$ using the global variable $r$. That is, $r$ is accessed within $p$.

The actual usage pair $AUP$ represents the number of times a module actually accesses a global data item. The potential usage pair $PUP$ represents the number of times a module could access a global variable. A potential usage of a variable $r$ by $p$ indicates that the scope of $r$ includes $p$.

The relative percentage of actual usage $RUP$ is then $RUP = AUP/PUP$. This formula provides a rough measure of the likelihood that an arbitrary segment will reference an arbitrary global variable. The greater the likelihood, the greater the possibility that a given global variable may have its value changed in another segment, without the knowledge of the maintenance programmer.

Table 1.
Operator count for actual implementation.

| OPERATOR | COUNT |
|---|---|
| BEGIN...END | 1 |
| ; | 11 |
| DO...END | 3 |
| = | 5 |
| < | 1 |
| TO | 2 |
| IF...THEN | 1 |
| ( ) | 6 |
| $n1 = 8$ | $N1 = 30$ |

Such an oversight may increase the chance of error when the software is modified.

For example, assume that we have a program with three global variables, $z1$, $z2$, and $z3$. The program has three subroutines, $A$, $B$, and $C$. If each subroutine has the global variables $z1$, $z2$, and $z3$. The program has three subroutines, $A$, $B$, and $C$. If each subroutine has the global variables $z1$, $z2$ and $z3$ available for its use, then, we have the following nine potential usage pairs:

| | | |
|---|---|---|
| $(a,z1)$ | $(b,z1)$ | $(c,z1)$ |
| $(a,z2)$ | $(b,z2)$ | $(c,z2)$ |
| $(a,z3)$ | $(b,z3)$ | $(c,z3)$ |

In this case, $PUP = 9$. Further, suppose subroutine $A$ actually references all three global variables, subroutine $B$ references two, and subroutine $C$ references none. Then, $AUP = 5$, and $RUP$ becomes $RUP = 5/9$.

Table 2.
Operand count for actual implementation.

| OPERAND | COUNT |
|---|---|
| I | 5 |
| N | 1 |
| J | 4 |
| X | 6 |
| SAVE | 2 |
| $n2 = 5$ | $N2 = 18$ |

Table 3.
Operator count for potential implementation.

| OPERATOR | COUNT |
|---|---|
| CALL | 1 |
| SORT( , ) | 1 |
| $n1^* = 2$ | $N1^* = 2$ |

Table 4.
Operand count for potential implementation.

| OPERAND | COUNT |
|---|---|
| X | 1 |
| N | 1 |
| $n2^* = 2$ | $N2^* = 2$ |

Table 5.
Software science parameters for bubble sort program.

| | |
|---|---|
| Unique Operators $N1$ | 8 |
| Unique Operands $N2$ | 5 |
| Total Operators $N1$ | 30 |
| Total Operands $N2$ | 18 |
| Vocabulary $N$ | 13 |
| Observed Length $N$ | 48 |
| Calculated Length $N'$ | $8 \log_2 8 + 5 \log_2 5 = 47$ |
| Volume $V$ | $48 \log_2 13 = 192$ |
| Potential Volume $V^*$ | $4 \log_2 4 = 8$ |
| Program Level $L$ | $12 \quad 192 = 0.063$ |
| Program Level $L'$ | $2 \quad 8 \times 5 \quad 18 = 0.069$ |
| Intelligence Content $I$ | $0.069 \times 192 = 13.2$ |
| Effort $E$ | $192 \quad 0.069 = 2782$ |

*Chapin's Q measure.* In this method,[14] data items are viewed differently, depending on how they are used, and data are divided into four categories:

- Role *P* data: input needed to produce a segment's output.
- Role *M* data: data that are changed or created within a segment.
- Role *C* data: data used in a "controlling" role within a segment.

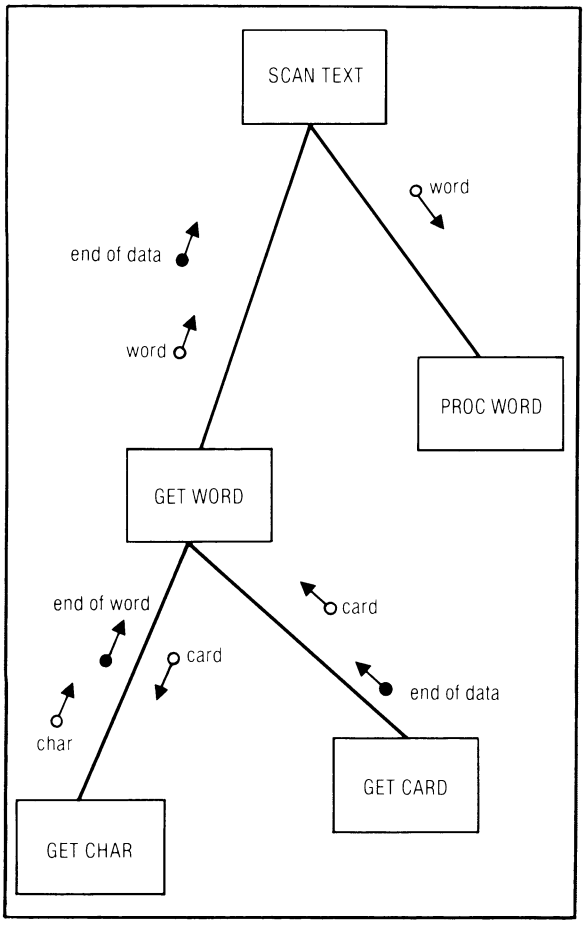



**Figure 3. Structure chart for sample program to illustrate Chapin's *Q* measure of complexity. Table 6 depicts the input-out data classification in the programs.**

Table 6.
Input-output table used in Chapin's *Q* measure of complexity.

| | INPUT | | | OUTPUT | | |
|---|---|---|---|---|---|---|
| SEGMENT | VAR | TYPE | FROM | VAR | TYPE | TO |
| SCAN TEXT | word | I | GET WORD | word | I | PROC WORD |
| | end of data | C | GET WORD | | | |
| GET WORD | char | P | GET CHAR | word | M | SCAN TEXT |
| | end of word | C | GET CHAR | end of data | I | SCAN TEXT |
| | card | I | GET CARD | card | I | GET CHAR |
| | end of data | CI | GET CARD | | | |
| GET CHAR | card | CP | GET CARD | card | M | GET WORD |
| | | | | end of data | I | |
| GET CARD | char | P | GET WORD | char | M | GET WORD |
| | | | | end of char | C | |
| PROC WORD | word | P | SCAN TEXT | none | | |

- Role *T* data: data that pass through a segment unchanged.

Since a particular datum can have different roles within different modules or even within the same module, it is counted as having each role.

Chapin observes that each type of data usage contributes different amounts of complexity to the module it is in. He notes that role *C* data contribute most to complexity, since they control which module will be invoked—that is, which course of action will be followed. Role *M* data contribute less than role *C* data but still provide a substantial amount of complexity, since their value is either initially defined or modified. Because role *P* data are often used to modify the value of role *M* data, role *P* data also contribute some complexity. Role *T* data, which have no effect on the module, contribute very little complexity.

An "input-output table" can be used to help classify each datum.[15] This table consists of a list of each input and output datum and its source (destination) for every module or segment in the program. The user computes program complexity *Q* by counting the number of data items used in *C, P,* or *T* roles for input and *M* or *T* roles for output and listing them in the input-output table. For each segment, the user multiplies the count by the appropriate weighting factor suggested by Chapin to account for the differing complexity contributed by each type of data. The weighting factor for role *C* data is 3; for role *M* data, 2; for role *P* data, 1; and for role *T* data, 0.50. These weighted products are then summed for each module, producing an intermediate measure $W'$.

The final measure takes into account the increase in complexity that is due to repetition factor *R*. This *R* value results from data being communicated between iteratively invoked segments and is calculated as follows. First determine which module contains exit tests for iteration that involve more than a single module. For every role *C* datum in these modules whose value comes from outside the loop body, add 2 to the iteration-exit factor *E,* which has an initial value of 0 for each segment. If the *C* role datum is created or modified in a segment other than the segment performing the exit test, but is still within range of the iteration, add 1 to *E*. The repetition factor *R* for each segment is then calculated by $R = (1/3 \times E)^2 + 1$. Note that if the segment does not perform an iteration-exit test, it has an *E* of 0, and hence an *R* of 1.

The index of complexity for each module *Q* is the square root of the sum of the weighted counts for that module $W'$ times its repetition factor *R*. That is, $Q = \sqrt{R \times W'}$.

The complexity for the entire program is then computed by calculating the arithmetic mean of the individual segment complexities. The entire process is illustrated in the following example.

We have a program consisting of five segments, whose relationships are shown in Figure 3, and whose input and output are shown in Table 6. The measures for each segment are

- SCAN TEXT $W' = 4$, $R = 1.1111$, $Q = 2.1082$
- GET WORD $W' = 10$, $R = 1.4445$, $Q = 3.8007$

- GET CARD $W' = 8$, $R = 1.0000$, $Q = 2.8240$
- GET CHAR $W' = 3.5$, $R = 1.0000$, $Q = 1.8708$
- PROC WORD $W' = 1$, $R = 1.0000$, $Q = 1.0000$

The total of $Q$ for all segments—that is, the overall program complexity—is 11.6037/5 or 2.3207.

Overall, data structure and flow metrics fail to be very comprehensive because the span between data references indirectly measures program length in some cases, but not consistently enough to qualify as a true hybrid.

Most of these techniques are also not widely applicable. For example, the segment-global usage pair is only useful with software that consists of programs that are segmented and use global data, and Chapin's $Q$ depends on the use of segments that communicate among themselves.

In addition, since the segment-global usage pair and the Chapin's $Q$ depend on data communicated between segments or modules, they clearly fail to be comprehensive even within the area of data complexity. While we can easily see that intermodule or global data can have a detrimental effect on program understandability, other properties of data can have similar effects—for example, the span between data references.

Finally, data structure and flow metrics have not been used in studies of their predictive power for software maintenance.

**Program control structures.** The majority of the work in software complexity over the past 10 years has dealt with the effects of control flow on program complexity. For example, a 50-line program with 25 IF-THEN-ELSE statements has well over 33 million possible control paths within the 50 statements.[16] Such a configuration can obviously be difficult to comprehend fully.

The complexity of control flow is commonly measured in density of control transfers within the program or in interrelations of control transfers.

Either approach to measuring control flow complexity normally represents a program as a flow graph to expose the control flow topology. The flow graph of a program is simply a directed graph that corresponds to the program's flow of control.

For example, directed graph $G = (V,E)$ consists of a set of nodes $V$ and a set of directed edges $E$ connecting the nodes. In a flow graph, each node represents a "sequential block of code," which is a sequence of instructions that can be entered only at the beginning of the sequence, can be exited only at the end of the sequence, and can contain no transfers of control within the sequence itself. The edges correspond to the flow of control between the various nodes.

In edge $(u,v)$ node $u$ is the initial node, and node $v$ is the terminal node. The number of edges that have a particular node $w$ as the initial node is the *outdegree* of $w$, and the number of edges that have $w$ as the terminal node is the *indegree* of $w$.

If an edge exists from some node $u$ to some node $v$, $u$ is said to immediately precede $v$, and $v$ is said to immediately succeed $u$. If a *path* exists from some node $u$ to some node $v$ consisting of one or more edges, $u$ precedes $v$, and $v$ succeeds $u$.

Since all the measures discussed in this section incorporate these concepts, we present Figure 4 to illustrate the theory behind program flow control. Note that Node $b$ has an indegree of 1 and an outdegree of 3. Further, node $b$ immediately precedes nodes $c$, $d$, and $e$ and immediately succeeds node $a$. Also, node $b$ precedes, in addition to nodes $c$, $d$, and $e$, nodes $g$, $h$, and $v$.

*McCabe's cyclomatic complexity.* McCabe[16] has proposed a graph-theoretic complexity measure that is widely accepted, probably because it is easily calculated and is intuitively satisfying. McCabe suggests that cyclomatic complexity is applicable in determining how difficult program testing will be, and empirical studies have been carried out on the effectiveness of this cyclomatic measure with favorable results.[8,9] McCabe's measure is based on the cyclomatic number $V(G)$ of a program's flow graph. For a flow graph with $e$ edges, $n$ nodes, and $p$ connected components (usually 1), the cyclomatic complexity is calculated using $V(G) = e - n + 2 \times p$.

The cyclomatic number may be viewed as the number of linearly independent circuits in a strongly connected graph, meaning that for any two nodes, one is reachable from the other. In other words, the cyclomatic number is the number of basic paths that can be combined to make up any possible circuit on the graph. The formula for calculating the cyclomatic complexity of the weakly connected flow graph in Figure 5 is $V(G) = 17 - 13 + 2 \times 1 = 6$.

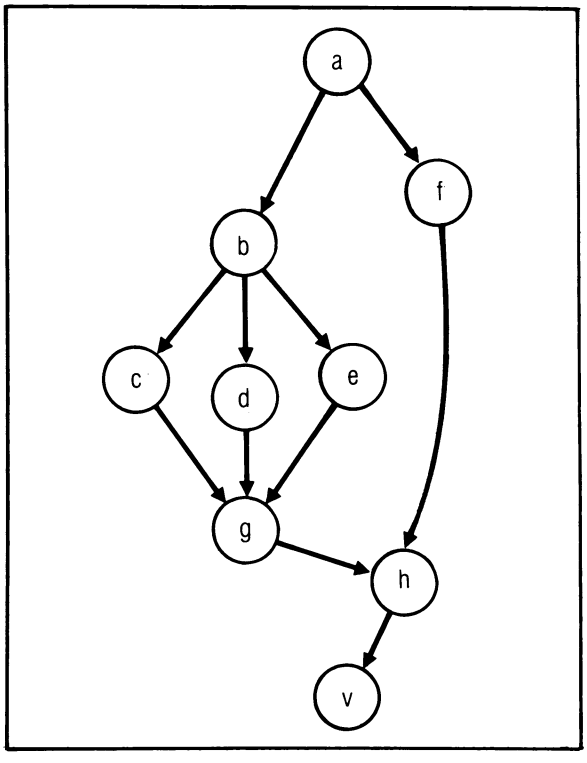



**Figure 4. Directed graph illustrating program control flow. Node *b* is the terminus of only one edge but is the beginning of three edges. It thus has an indegree of 1 and an outdegree of 3.**

Further, if we convert the flow graph into a strongly connected graph by adding an edge from the terminal node *m* to the initial node *a* (Figure 6), we increase the cyclomatic number by 1. With the new edge, we are adding an additional "implicit" decision—to follow the edge back to node *a* or to end the path at *m*. The strongly connected flow graph in Figure 6 contains the following seven independent circuits:

$$a-b-d-a$$
$$a-b-d-h$$
$$a-b-e-h-j$$
$$a-c-g-i-j$$
$$a-c-f-i-j$$
$$j-k-m-a$$
$$j-l-m-a$$

*Myers' extension to the cyclomatic number.* Myers[17] extends McCabe's theories by noting that predicates with compound conditions are more complex than predicates with a single condition. The following code segments illustrate this point:

IF $x=0$ THEN $s1$ ELSE $s2$

IF $x=0$ and $y=1$ THEN $s1$ ELSE $s2$

Since both segments involve only a single decision, they can both be illustrated by the same directed graph (Figure 7), which has a cyclomatic complexity $V(G)$ of 2. However, their predicates differ in complexity.

Myers suggests that by calculating the complexity measure as an interval rather than as a single value, we can get a more accurate complexity measure. In his approach, the interval's lower bound is the number of decision statements plus 1, as in McCabe's $V(G)$, and the upper bound is the number of *individual* conditions plus 1.

In this manner, Myers hoped to construct a complexity measure that would account for both the decision statements themselves and their predicates.

Using Myers' scheme, the first code segment has an associated interval [2,2], and the second segment has [2,3] to allow a finer distinction between programs with similar flow graphs.

Unfortunately, no results comparing Myers' measure to maintenance difficulties have been published, so their applicability is uncertain.

*Gilb's logical complexity metric.* Gilb[18] defines logical complexity as a measure of how much decision-making logic is in the program. He proposes two measures: *CL*, or absolute logical complexity, which is the number of "binary" decisions in the program's logic and *cl*, or relative logical complexity, which is the ratio of absolute logical complexity to the total number of statements in the program.

Experiments support Gilb's assumption that the degree of decision-making logic in the program can be correlated

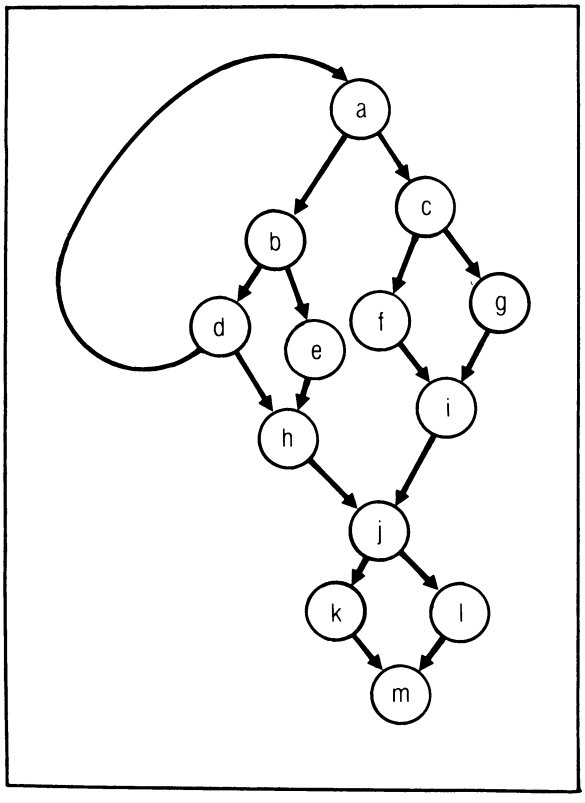



**Figure 5. Weakly connected flow graph. In this graph of program flow, not all nodes are reachable from other nodes (*m* to *a*, for example). With some adjustment, however, this graph can become strongly connected.**

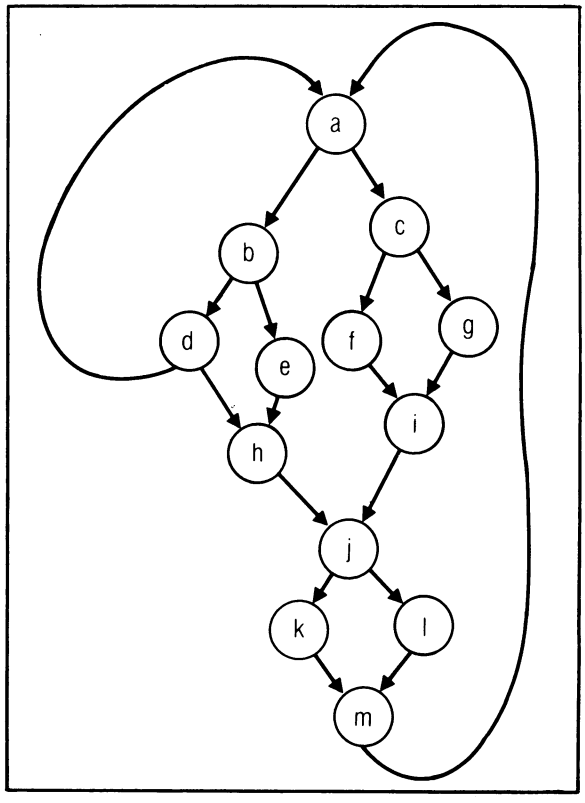



**Figure 6. Strongly connected flow graph. The weakly connected program flow graph in Figure 5 has been adjusted by adding an edge from node *m* to node *a*. The result is that all nodes are reachable from other nodes.**

to characteristics of that program such as error proneness, development cost, and development time. This general notion is suppported by Sime et al.[19,20] who have shown that various forms of conditional constructs that "simplify" control somewhat (IF-THEN-ELSE versus IF-THEN-GO TO) favorably affect programming time and the number of errors. Farr and Zagorski[21] have also found that the degree of decision-making within a program is a significant factor in predicting software costs.

Gilb sees such a measurement as an indirect tool for analyzing and perhaps controlling the program characteristics mentioned above.

*The knot count of Woodward, Hennell, and Hedley.* Woodward, Hennell, and Hedley[22] suggest a method in which they examine the relations between the physical locations of control transfers rather than simply their numbers. This method can be easily calculated as follows.

Let transfer of control from line $a$ to line $b$ be the ordered pair $(a,b)$, with min $(a,b)$ referring to the first line of the pair $(a,b)$ and max$(a,b)$ referring to the last line. Under these assumptions, a "knot" occurs when $\min(a,b) < \min(p,q) < \max(a,b)$ and $\max(p,q) > \max(a,b)$, or when $\min(a,b) < \max(p,q) < \max(a,b)$ and $\min(p,q) < \min(a,b)$. In other words, a knot occurs when we "jump" out of the scope of the $(a,b)$ transfer (Figure 8).

This technique may be generalized to apply to program flow graphs by letting the ordered pair of line numbers represent an edge of the graph. Hence the pair $(a,b)$ would now refer to the edge with initial node $a$ and terminal node $b$, rather than a transfer from line $a$ to line $b$ in the program.

Difficulties can arise in obtaining an exact knot count, since the node in a flow graph actually refers to a sequence of instructions that may be entered into only at the beginning and exited from only at the end, with no internal transfers of control. The problem is shown more clearly in Figures 9 and 10. In Figure 9, line $c$ makes up $v_3$ while line $d$ is $v_4$. In Figure 10, both lines $c$ and $d$ make up node $v_3$. Therefore, we can say that if the node includes only one line, a knot is present, but if it contains more than one, a knot is not present.

An interval can be used to get a more precise knot count, with the lower bound being the verifiable number of knots detected and the upper bound the total number of possible knots, assuming every block contains one statement. Neither the interval nor the basic knot count has been applied to the maintenance of real programs, however.

*Chen's measure of program complexity.* Chen[23] has developed a topological complexity measure that is sen-

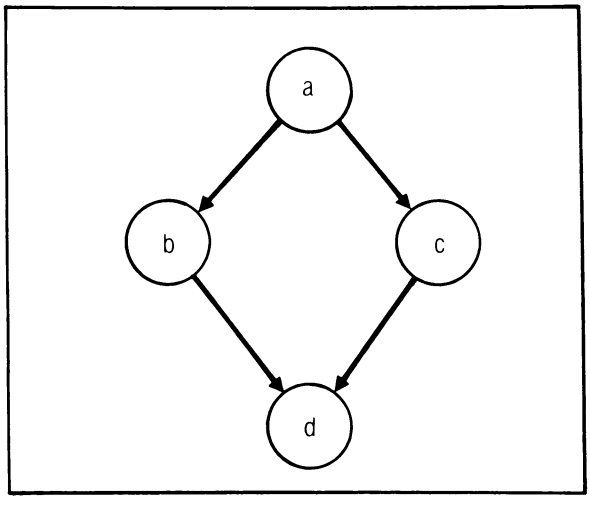



**Figure 7. Flow graph of a one-decision code segment.**

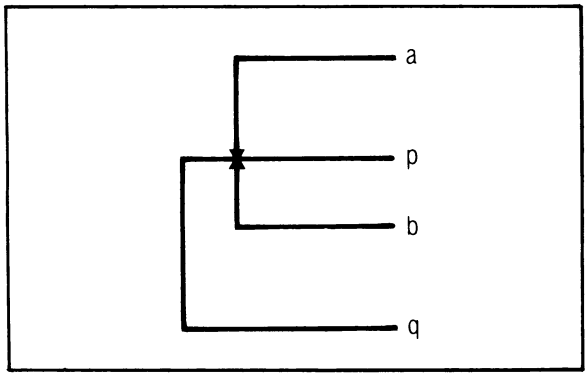



**Figure 8. A control flow knot occurs when the transfer from *a* to *b* is interrupted, and some other transfer outside the *a*-to-*b* scope is required.**

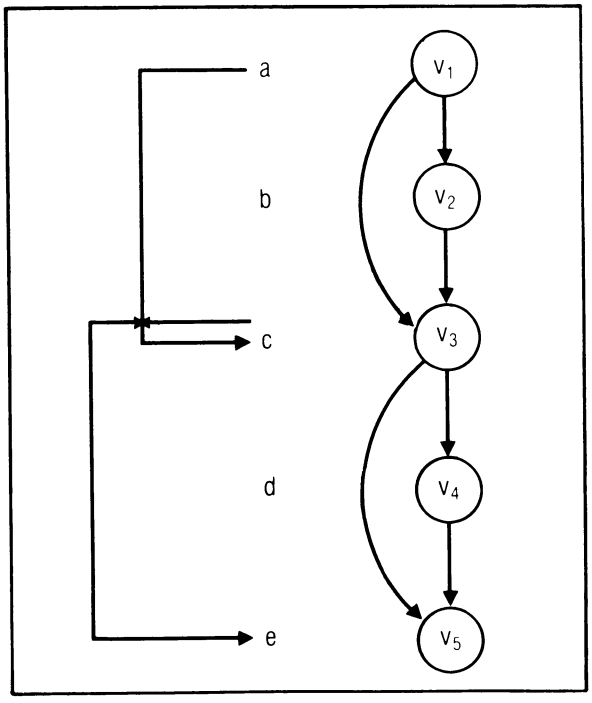



**Figure 9. Flow graph with knot. A knot occurs when transfer between two nodes is interrupted. Each node is a sequence of events that has one entrance and one exit with no internal transfers. Therefore, since a one-line-per-code construction requires an internal transfer, a knot results.**

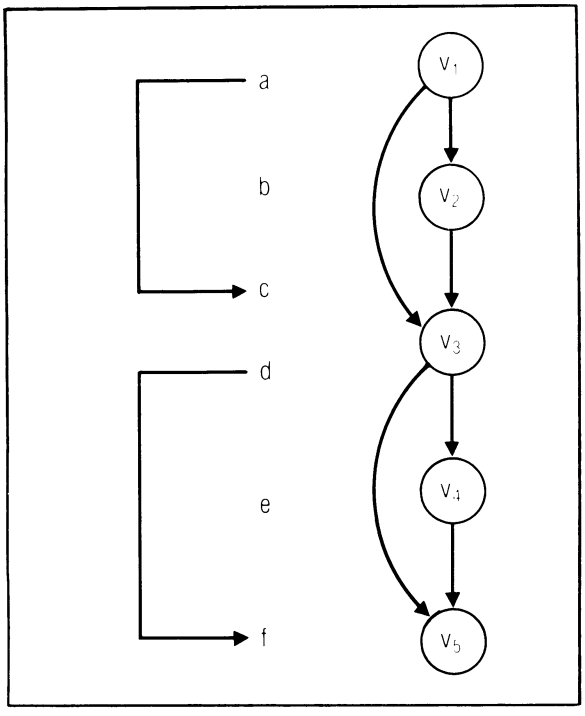



Figure 10. Flow graph without knot. Node $v_3$ has two lines instead of the one line in Figure 10. Since no internal transfer is needed, no knot occurs.

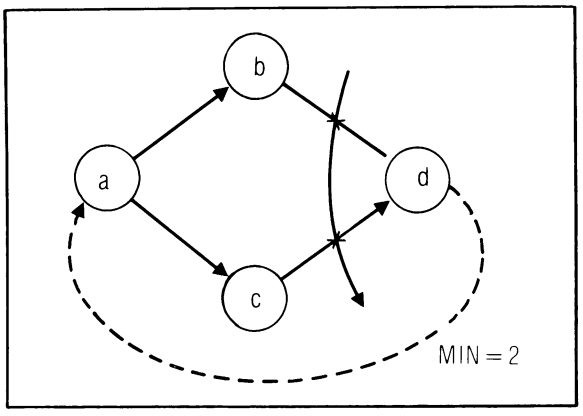



Figure 11. Illustration of maximal intersect number, or *min*. The *min* is derived by first determining the number of regions possible, (edges – nodes) + 2, and then drawing a line so that it enters each region exactly once. The number of times the line intersects the edges is the *min*.

sitive to nested decision structures. His technique uses the maximal intersect number *min* of the program's flow graph.

To compute the *min* the flow graph must be converted (if it is not already) into a strongly connected graph by connecting the terminal and initial nodes with an edge. Such a graph divides the two-dimensional space that it occupies into a finite number of regions. (McCabe showed that the number of regions in a connected, planar graph without bridges, or edges that disturb the weak connectivity of the graph, is equal to the cyclomatic number of that graph by rearranging Euler's theorem from $n-e+r=2$ to $r=e-n+2$.)

The *min*, then, is the number of times a line intersects the edges of the graph when the line is drawn such that it enters every region of the graph exactly one time (Figure 11).

The *min* may be calculated in this manner only for graphs that do not have bridges. If a graph does have a bridge, then the *min* for that graph is calculated by summing the *min*s of the strong components (the maximal subgraphs that do not contain bridges), subtracting twice the number of strong components, and adding 2 (Figure 12).

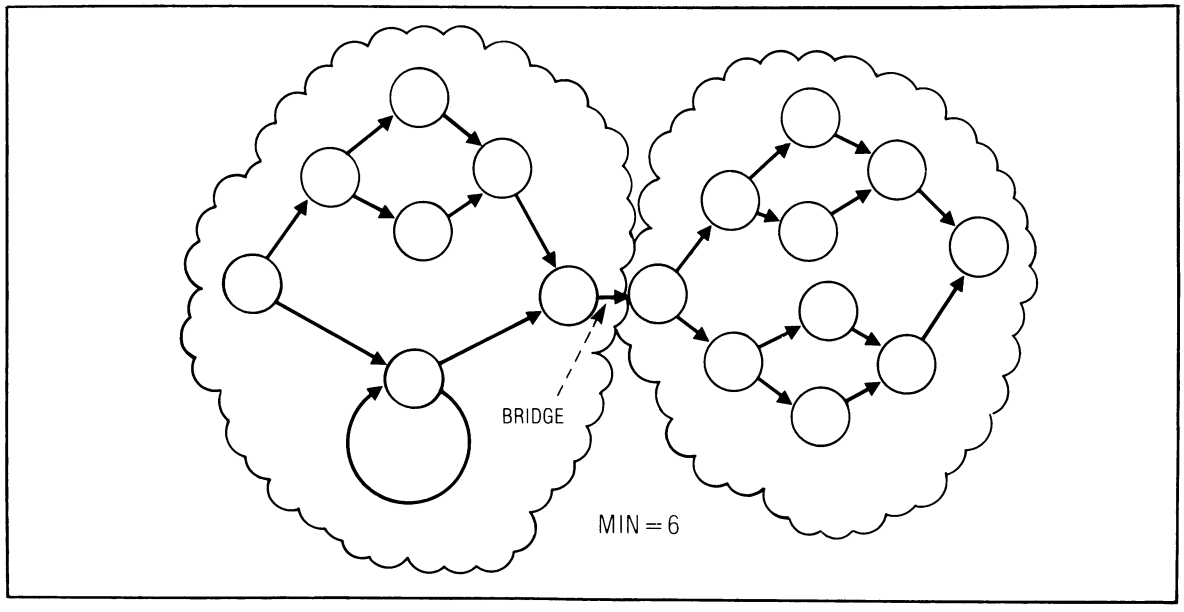



Figure 12. Illustration of maximal intersect number for graph with a bridge. Here we have two subgraphs connected by a bridge. The *min* of the first is 4, as is the *min* of the second. By adding the two *min*s, subtracting twice the number of subgraphs (4), and adding 2, we get a *min* of 6.

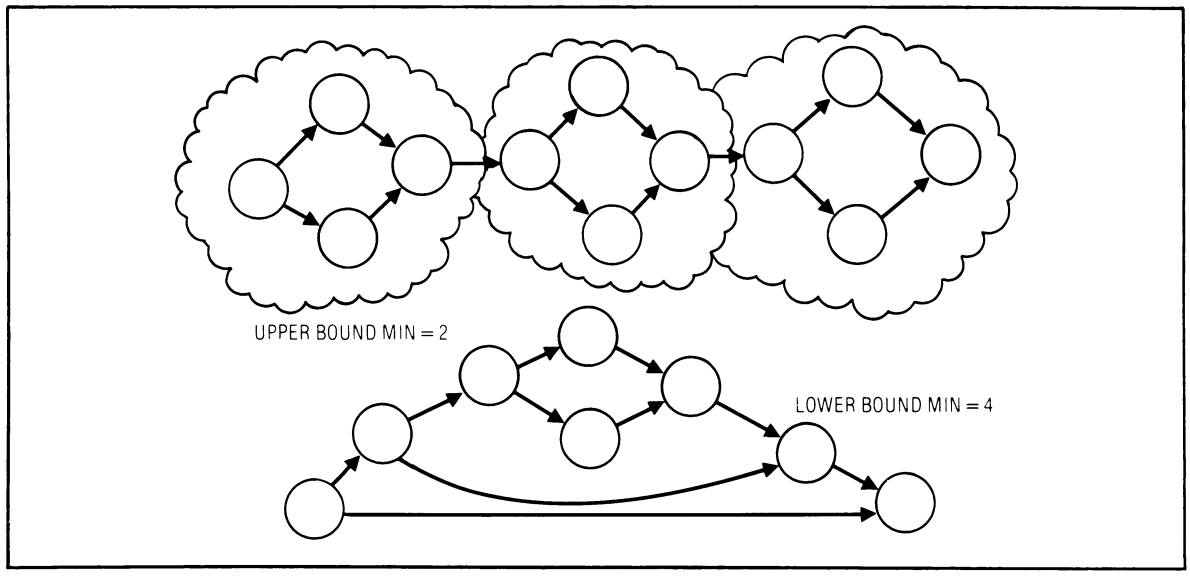



**Figure 13. Illustration of upper and lower bounds of maximal intersect number. Note that the upper bound contains nested decisions, while the lower bound decisions are all serial.**

In general, the upper bound of a given flow graph's *min* is $n+1$, where the flow graph has $n$ decisions, and the lower bound is 2. The upper bound occurs when the flow of control is arranged so that every decision is nested. The lower bound is obtained when every decision is serial—that is, unnested (Figure 13).

*The scope metric.* The scope metric,[24,25] like other metrics that measure control flow complexity, is based on the graph-theoretic representation of a computer program as a flow graph.

Let a computer program be represented as flow graph $G=(V,E)$ with a single initial node and a single terminal node. Then, the set of nodes $V$ can be partitioned into two sets—those with an outdegree of one or less, called *receiving* nodes and those with an outdegree of two or more, called *selection* nodes.

To obtain the scope measure of complexity, we create a subgraph $G'$, consisting of all nodes that immediately succeed a given selection node. That is, the subgraph consists of all nodes that are connected by a single edge from the selection node (including the selection node itself, in a self-loop).

Each of these subgraphs has at least one node in the graph $G$ that is its lower bound. A lower bound is a node that can be reached from every node in the subgraph; that is, it succeeds every node that immediately succeeds the selection node. Most subgraphs will have a number of lower bounds, but the lower bound that precedes every other lower bound of the subgraph is called the *greatest lower bound.*

The number of nodes preceding the greatest lower bound of a selection node's subgraph (excluding the GLB itself) and succeeding the selection node, plus 1, yields the *adjusted complexity* for that selection node. Each receiving node has an adjusted complexity of 1, except for the terminal node, which has an adjusted complexity of 0. The adjusted complexity of each node, both selection and receiving nodes, is summed to get the overall complexity of the flow graph.

The process of computing the scope measure of complexity for the flow graph in Figure 14 is illustrated in Tables 7 and 8.

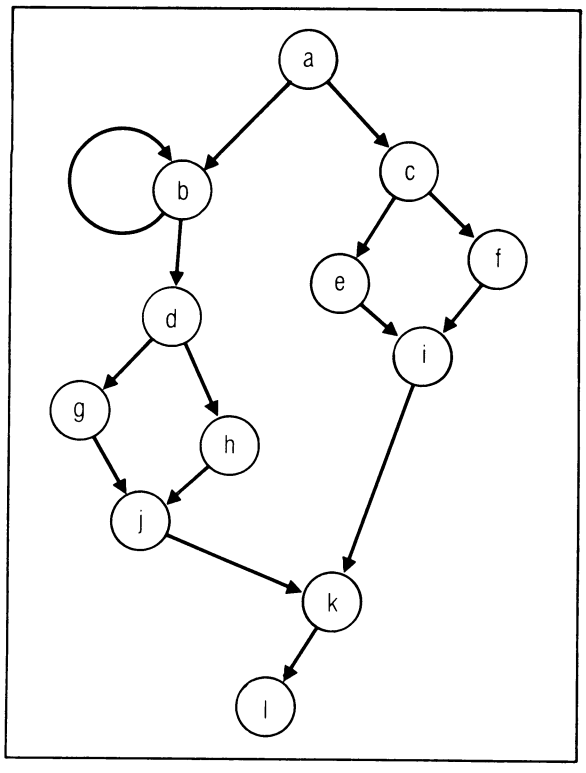



**Figure 14. Flow graph used in scope complexity calculations. Subgraphs are explained in Table 7, and scope complexity is computed in Table 8.**

Table 7.
Subgraphs within flow graph in Figure 14.

| | | | | |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |

Table 8.
Computation of scope complexity of flow graph in Figure 14.

| NODE | COMPLEXITY |
|---|---|
| | 10 |
| | 2 |
| | 3 |
| | 3 |
| | 1 |
| | 1 |
| | 1 |
| | 1 |
| | 1 |
| | 1 |
| | 0 |
| TOTAL | 25 |

*The scope ratio.* The scope number, in essence, is the number of nodes in the flow graph. Obviously, this measure can not always be reliable, since some programs can be trivially rearranged to give flow graphs with different scope measures, as shown in Figure 15. For this reason, the scope ratio was developed.[26]

The scope ratio is calculated by dividing the number of nodes in the flow graph, excluding the terminal node, by the scope number. In this case, as the ratio increases, the complexity decreases, but the scope ratio can be taken as one minus the original ratio. This number is much more satisfactory, since it increases towards one as complexity increases, and decreases to zero as complexity decreases.

If the scope ratio is used to analyze the flow graph in Figure 14, it would yield a value of $1 - (11/25)$, or 0.56.

The advantage of the scope measure over current forms of control flow metrics is its sensitivity to nested decisions, which is illustrated in the complexity measures of the flow graphs in Figure 16. The results seem to satisfy the rankings we would intuitively assign to these flow graphs.

As with every other measure yet surveyed, the control flow metrics fail to be comprehensive. They do not take into account the contribution of any factor except control flow complexity.

However, control flow metrics do a fairly good job of differentiating between two programs that are otherwise equivalent in other characteristics such as size. A useful approach may be to use control flow metrics to differentiate among programs that have already been placed in the same size categories using size metrics.

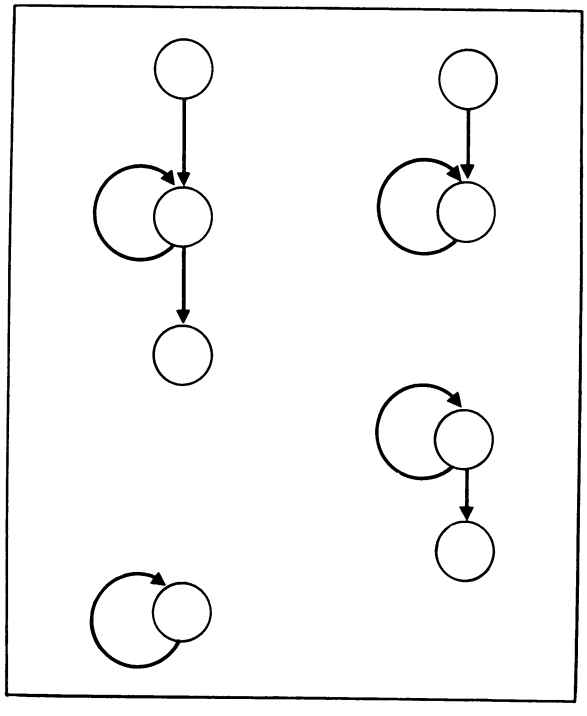


**Figure 15. Possible one-decision flow graphs. Although all these graphs depict one decision, the number of nodes involved, and hence the scope measure, is not the same for each.**

*Hybrid complexity measures.* Hybrid complexity measures attempt to remedy one of the shortcomings of the single-factor complexity metrics in use. They consider two or more properties that are thought to contribute to software complexity—program size, program data structures, and program flow of control.

One approach is to borrow part of the measure from an existing metric, such as Hansen's,[27] which combines a measure of control flow and program size. An alternative approach is to develop completely new measures of complexity for the various properties and combine them, as does Oviedo.[28]

*Hansen's measure of complexity.* Hansen[27] developed a measure that combines the cyclomatic complexity of McCabe and a count of operators similar to Halstead's $n1$.

Hansen's complexity measure consists of a 2-tuple $(a,b)$, where the first value is a count of the number of

- IF, CASE, or other alternate execution constructs and
- iterative DO, DO-WHILE or other repetitive constructs.

The second component of the 2-tuple is a count of operators in the program, which Hansen defines as

- primitive operators such as +, −, *, AND, SUBSTR, etc;
- assignment;
- subroutine or function calls;
- application of subscripts to an array; and
- input and output statements.

Hansen suggests that some version of the cyclomatic number is an appropriate measure of control flow complexity, since it is easy to compute and supports the use of an operator count by observing that a program with more operators is bigger; hence "more is going on" within the program. Because each operation must be understood to understand the whole, a bigger program is more complex.

Hansen has demonstrated this technique by applying it to four programs and their revisions, which were pub-

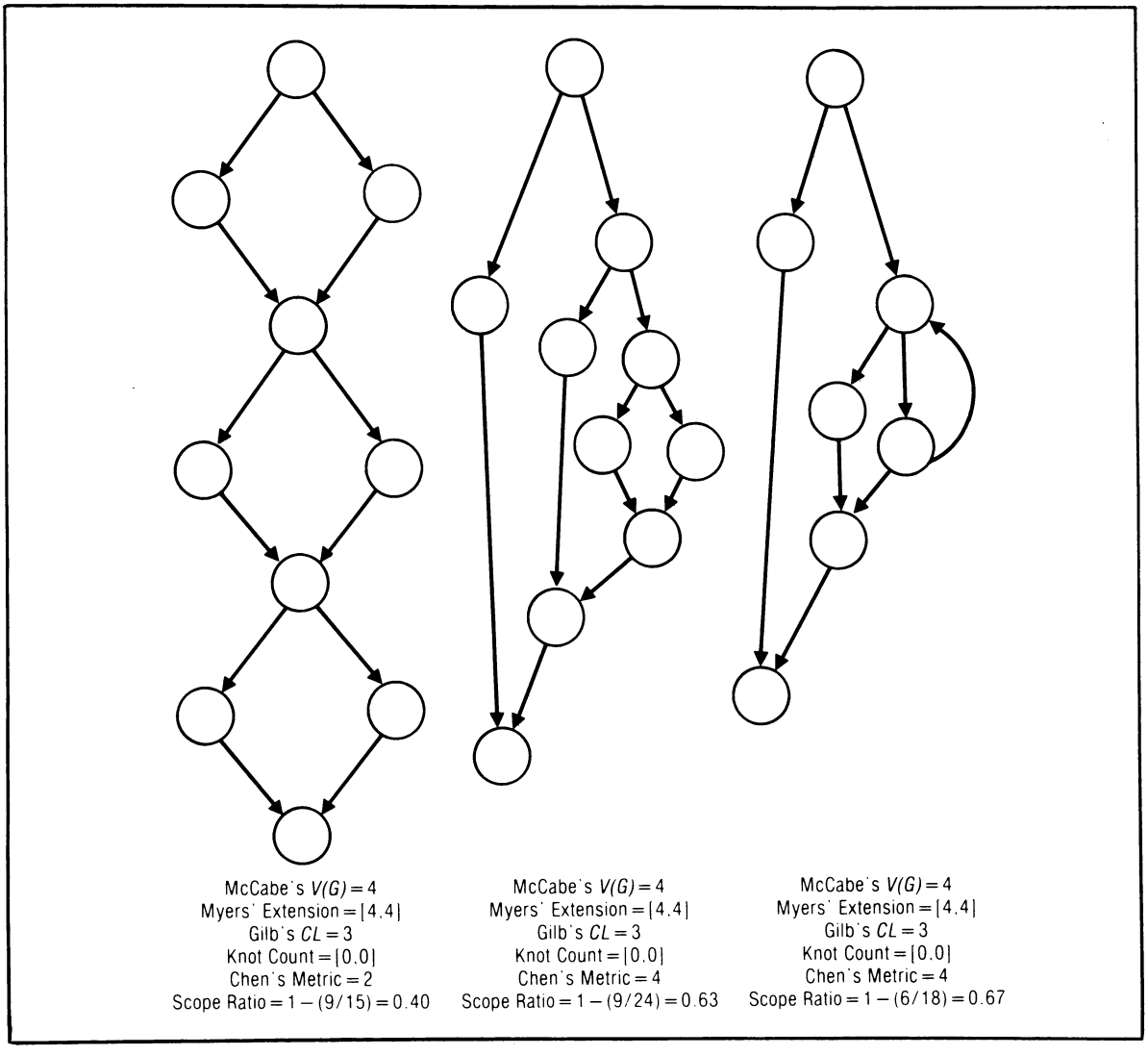



**Figure 16. Flow graphs with three decisions.**

lished in a popular programming style text by Kernighan and Plauger.[29] In each case, the complexity of the original program's 2-tuple indicated greater complexity than the revised version's 2-tuple, which was written to conform with the text's guidelines on programming style.

Few empirical studies can support Hansen's measure. However, the components of the measure have been independently validated in their original form, which would tend to lend a measure of credibility to this metric.

*Oviedo's model of program complexity.* Oviedo[28] has developed a method that measures data flow complexity and control flow complexity. First, control flow complexity $cf$ is calculated, and then data flow complexity $df$ is calculated. Total program complexity $C$ is then $C = acf + bdf$, where $a$ and $b$ are appropriate weighting factors. In his preliminary work, Oviedo suggests that the weighting factors be $a = b = 1$ until further experimentation can be done.

The control flow complexity $cf$ is easily calculated once the program is represented as a flow graph—it is simply the number of edges in the graph. The calculation of the data flow complexity is a bit more complicated, however,

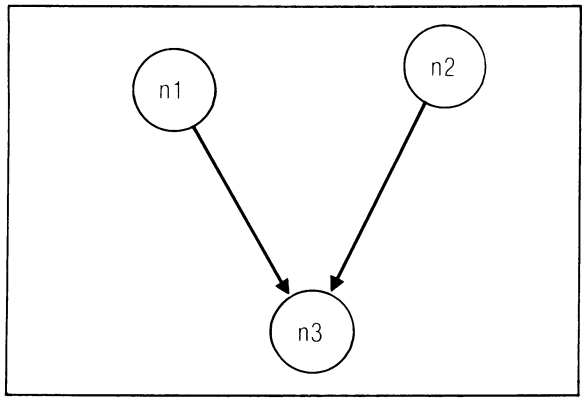



**Figure 17. Illustration of locally exposed variables that can reach n3.**

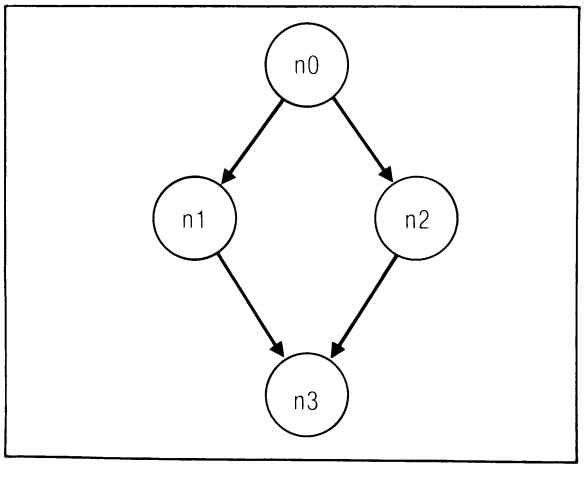



**Figure 18. Flow graph of sample program to illustrate Oviedo's complexity model.**

and requires some preliminary definitions. A *variable definition* occurs when a variable is assigned a value, either through an input statement, an assignment statement, or a function or subroutine call. A *variable reference* occurs when the value of a variable is used, either in an output statement or some sort of expression such as an assignment statement.

In a definition-reference relation, the definition and reference occur in the same node or they occur in different nodes. Allen and Cocke[30] term a variable *locally available* for a block if there is a definition of that variable within the block. A variable is termed *locally exposed* if it is referenced in a node and is not preceded in that node by a definition of that variable.

A variable definition in node $n_i$ can reach a block $nk$ if there is a path from $n_i$ to $nk$ such that the variable is not locally available in any node on the path. A variable definition kills all other definitions of the variable when the definition is encountered.

The data flow complexity of node $n_i$, called $df_i$, is the number of prior definitions of locally exposed variables in $n_i$ that can reach $n_i$ (Figure 17). Assume that $n1$ of Figure 17 consists of $x: = 1, j: = 2$, and $m: = 5$. Further, let node $n2$ consist of $k: = 1$ and $j: = 3$, and node $n3$ consist of $d: = x + j + k$. Then, the locally exposed variables in node $n3$ are $x$, $j$, and $k$. Note that variables $x$ and $k$ were previously defined one time each and $j$ twice, making $df$ equal to 4.

Data flow complexity for the program is then equal to the sum of the data flow complexity of each node. That is,

$$df = \sum_{i=1}^{v} df_i$$

Note that the data flow complexity of the first node $df1$ is always zero. The total program complexity is

$$c = e + \sum_{i=1}^{v} df_i$$

For example, let $n0$ be the initial node of a program, and nodes $n1$, $n2$, and $n3$ be defined as above. The flow graph of such a program may appear similar to the flow graph in Figure 18 and the program itself might be

```
READ n,x,k
If n = 1 THEN
        x: = 1
        j: = 2
        m: = 5
ELSE
        k: = 1
        j: = 3
ENDIF
d: = x+j+k
```

Notice the addition of the level statement and the IF-THEN-ELSE construct to the previous program. The complexity of node $n0$ that consists of

READ $n,x,k$
IF $n=1$

is zero, since no prior definitions reach this block. Nodes $n1$ and $n2$ have no locally exposed variables, since they consist only of assignment statements, which use constant values. Node $n3$ has three locally exposed variables, $x$, $j$, and $k$. For each locally exposed variable the following number of definitions reach $n3$:

| | |
|---|---|
| $x$ | 2 |
| $j$ | 2 |
| $k$ | 2 |
| | 6 |

Therefore node $n3$ has a data flow complexity of 6, or $df3=6$. Then $df=6$, since $df0=0$, $df1=0$, $df2=0$ and $df3=6$. This sample clearly has four edges, so $|e|=4$. If $a=b=1$, then $c=4+6=10$. Hence, the sample program has an Oviedo complexity of 10.

*Integrating software science with the scope measure.* To develop a reliable metric that can assign a realistic complexity measure to a given computer program, components must be included that evaluate the other properties contributing to program complexity. A modification to the original scope measure was suggested for this reason,[31] and because it is the most context-sensitive of the program size metrics, software science[4] was selected as part of the modification. As we discussed earlier, software science is used to reflect the complexity contributed by program length and data.

We consider the program as flow graph $G=(V,E)$ and each node has a complexity assigned to it, which is called *raw complexity*.

The raw complexity of node $v_i$ is $e_i$, the software science measure of effort. The first node $v1$ has raw complexity $e1$, the second node $v2$ has raw complexity $e2$, etc. This measure is calculated in the following manner. The unique operand and operator parameters $n1$ and $n2$ are determined for the entire program—that is, the parameters are *global*. The total use parameters $n1$ and $n2$ for the operator and operand are determined for individual node or block $v_i$; that is, the parameters are *local*. This results in parameters $n1_i$, $n2_i$, and $N_i$ for the $i$th node. The computation of total $e$ values $E_i$ is $E_i=(N_i \log 2\, n)/L'$, where $L'=2/n1\times n2/N2$. Note that $L'$ is a global parameter.

The software science $e$ value for each node is used to compute adjusted complexities for the selection nodes. The adjusted complexity for a selection node is the sum of the $e$ values of every node within the "scope" of that selection node, plus the $e$ value of the selection node itself. A node is within the scope of a selection node if it (1) precedes the greatest lower bound of the subgraph consisting of all nodes that immediately succeed the selection node and (2) succeeds the selection node.

Receiving nodes (those with an outdegree of 1) have an adjusted complexity equal to their raw complexity. The

complexity of the overall program is the sum of the adjusted compexities of every node in the flow graph.

The hybrid approach to measuring software complexity is clearly the most sensible approach. Software complexity is caused by so many different factors that measuring only one of them cannot help but give unreliable results for a general case.

Since the hybrids presented here are not supported by a great deal of empirical research, their veracity must be evaluated by examining their components.

With Hansen's measure, the component that measures flow of control is similar to Gilb's *CL,* absolute logical complexity. A moderate amount of empirical work supports the utility of measuring the flow of control to determine complexity, and the operator count part of the metric is supported in part by work done to validate Halstead's software science.

However, if Hansen's measure shares some of their support, it must share all of their drawbacks. Gilb's measure lacks context sensitivity, and Halstead's work suffers from a similar problem. However, they are widely applicable to many types of software. Hansen also uses a 2-tuple, which creates problems with two measures such as (5,20) and (3,65), since determining which of the two is more complex is difficult. A single component measure of complexity is much more desirable. Oviedo's model, which measures the number of edges in a flow graph, is similar in principle to McCabe's and Gilb's measures and, like these two, fails to consider the context of each edge. Also, determining an appropriate weighting factor *a* for control flow complexity may be difficult. The second component, data flow complexity, seems to be intuitively satisfying, but no empirical studies have been reported to support this property's effect on program complexity. The weighting factor *b* for this component may also be difficult to determine.

The measures just presented represent the metrics being used in larger software-oriented companies and some universities. Some appear quite reliable and valid, but all have some problems associated with them. Each metric included can be implemented with a computer program, and while there may be some disagreement on what is included in the calculations (for example, lines of code and Halstead's software science), all can be said to be deterministic.

Table 9.
Usefulness of complexity metrics.*

| | EMPIRICAL EVIDENCE | CONTEXT SENSITIVE | WIDELY APPLICABLE | COMPREHENSIVE |
|---|---|---|---|---|
| LINES OF CODE | 2 | 1 | 3 | 1 |
| SOFTWARE SCIENCE | 3 | 1 | 3 | 2 |
| SPAN BETWEEN DATA REFERENCES | 1 | 1 | 3 | 2 |
| SEGMENT GLOBAL USAGE PAIR | 1 | 1 | 1 | 1 |
| CHAPIN'S Q MEASURE | 1 | 2 | 1 | 1 |
| CYCLOMATIC NUMBER | 3 | 1 | 3 | 1 |
| MYERS EXTENSION | 1 | 1 | 3 | 2 |
| GILB'S CL | 2 | 1 | 3 | 1 |
| GILB'S CL | 1 | 1 | 3 | 2 |
| KNOT COUNT | 1 | 1 | 2 | 1 |
| CHEN'S METRIC | 1 | 2 | 3 | 1 |
| HANSEN'S METRIC | 1 | 1 | 3 | 2 |
| OVIEDO'S METRIC | 1 | 1 | 3 | 2 |

* DOES NOT INCLUDE HALSTEAD

## Acknowledgment

Work by Harrison and Magel was supported in part by National Science Foundation Grant MCS 8002667.

## References

1. J. C. Baird and E. Noma, *Fundamentals of Scaling and Psychophysics,* John Wiley & Sons, New York, 1978, pp. 1-6.
2. V. Basili and A. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Trans. Software Eng.,* Vol. SE-1, Dec. 1975, pp. 390-396.
3. J. Elshoff, "An Analysis of Some Commercial PL/I Programs," *IEEE Trans. Software Eng.,* Vol. SE-2, June 1976, pp. 113-120.
4. M. Halstead, *Elements of Software Science,* Elsevier North-Holland, New York, 1977.
5. R. Bohrer, "Halstead's Criteria and Statistical Algorithms," *Proc. Eighth Ann. Computer Science Statistics Symp.,* Los Angeles, February 1975, pp. 262-266.
6. J. Elshoff, "Measuring Commercial PL/I Programs Using Halstead's Criteria," *ACM SIGPLAN Notices,* May 1976, pp. 38-46. (also GM Research Publication GMR-2012, 1975).
7. Y. Funami and M. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data," *Proc. Symp. Computer Software Eng.,* 1976, pp. 133-138.
8. B. Curtis et al., "Measuring the Psychological Complexity of Software Maintenance Tasks With the Halstead and McCabe Metrics," *IEEE Trans. Software Eng.,* Vol. SE-5, Mar. 1979, pp. 96-104.
9. B. Curtis, S. Sheppard, and P. Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," *Proc. Fourth Int'l Conf. Software Eng.,* 1979, pp. 356-360.

10. A. Feuer and E. Fowlkes, "Some Results from an Empirical Study of Computer Software," *Proc. Fourth Int'l. Conf. Software Eng.*, 1979, pp. 351-355.

11. S. Sheppard, P. Milliman, and B. Curtis, *Experimental Evaluation of On-Line Program Construction,* Tech Report TR-79-388100-6, Arlington, VA, GE Information Systems Programs, 1979.

12. A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science," *Computing Surveys,* Vol. 10, No. 1, Mar. 1978, pp. 3-18.

13. V. Basili, "Product Metrics," *Tutorial on Models and Metrics for Software Management and Engineering,* IEEE Computer Society Press, 1980, pp. 214-217.

14. N. Chapin, "A Measure of Software Complexity," *Proc. NCC,* 1979, pp. 995-1002.

15. N. Chapin, "Input-Output Tables in Structured Design," *Structured Analysis and Design, Volume 2,* Infotech Int'l., Ltd., Maidenhead, UK, 1978, pp. 43-55.

16. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, Vol. SE-2, Dec. 1976, pp. 308-320.

17. G. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," *ACM SIGPLAN Notices,* Oct. 1977, pp. 61-64.

18. T. Gilb, *Software Metrics,* Winthrop Publishers, Cambridge, MA, 1977.

19. M. Sime, T. Green, and D. Guest, "Psychological Evaluation of Two Conditional Constructions Used in Computer Languages," *Int'l J. Man-Machine Studies,* Vol. 5, No. 1, 1973, pp. 105-113.

20. M. Sime, T. Green, and D. Guest, "Scope Marking in Computer Conditionals—a Psychological Evaluation," *Int'l J. Man-Machine Studies,* Vol. 9, No. 1, 1977, pp. 107-118.

21. L. Farr and H. Zagorski, "Quantitative Analysis of Programming Cost Factors: a Progress Report" in "Economics of Automatic Data Processing," *Proc. ICC Symp.*, Frielink, ed., The Netherlands, 1965.

22. M. Woodward, M. Hennell, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," *IEEE Trans. Software Eng.*, Vol. SE-5, Jan. 1979, pp. 45-50.

23. E. Chen, "Program Complexity and Programmer Productivity," *IEEE Trans. Software Eng.*, Vol. SE-4, May 1978, pp. 187-194.

24. W. Harrison and K. Magel, "A Complexity Measure Based on Nesting Level," *ACM SIGPLAN Notices,* Mar. 1981, pp. 63-74.

25. W. Harrison and K. Magel, "A Graph-Theoretic Complexity Measure," *ACM Computer Science Conf.*, St. Louis, Missouri, Feb. 1981.

26. W. Harrison and K. Magel, "A Topological Analysis of Computer Programs With Less Than Three Binary Branches," *ACM SIGPLAN Notices,* Apr. 1981, pp. 51-63.

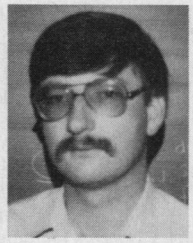
27. W. Hansen, "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)," *ACM SIGPLAN Notices,* Mar. 1978, pp. 29-33.

28. E. Oviedo, "Control Flow, Data Flow and Program Complexity," *Proc. COMPSAC 80,* pp. 146-152.

29. B. Kernighan and P. Plauger, *The Elements of Programming Style,* McGraw-Hill, New York, 1974.

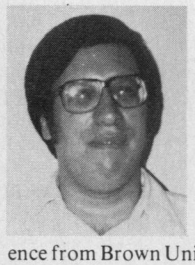
30. F. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Comm. ACM,* Vol. 19, No. 3, Mar. 1976, pp. 253-261.

31. W. Harrison, "A Hybrid Metric to Measure Software Complexity," MS thesis, University of Missouri-Rolla, 1981.



**Warren Harrison** is a doctoral student in computer science at the University of Oregon. He has a BS in accounting and information systems from the University of Nevada-Reno and an MS in computer science from the University of Missouri-Rolla. In 1980, he was awarded the Certificate in Data Processing.

Harrison has been a programmer with the Nevada Cooperative Extension Service, a computer scientist with Lawrence Livermore National Laboratory and a member of the technical staff at Bell Laboratories. His research interests include software maintenance, program testing, human factors, and decision support systems.



**Kenneth Magel** is an associate professor of computer science at the University of Texas at San Antonio. He taught for two years at Wichita State University and then for four years at the University of Missouri-Rolla. His research interests include software metrics, predictive software development tools, and the use of comprehensive models in program optimization.

Magel received a PhD in computer science from Brown University in 1977. He is a member of the IEEE Computer Society and the ACM.

**Raymond Kluczny** is an associate professor of engineering management at the University of Missouri-Rolla, where he has been since 1979. He received a DBA from Arizona State University in 1979. Kluczny's research interests are management information systems and systems analysis and design. He is a member of the Society for Management Information Systems and the ACM.

**Arlan DeKock** is professor and chairman of the Computer Science Department at the University of Missouri-Rolla, where he has taught since 1968. Previously, he worked for NASA, was database administrator for the Missouri Department of Social Services, and consulted for numerous private and governmental organizations. His research interests include software engineering, database design, and artificial intelligence. DeKock received his PhD in 1968 in Human Factors Engineering from the University of South Dakota. He is a member of the ACM.